

# Programación Funcional Avanzada

## Monads – Writer, State, IO y Reader

Ernesto Hernández-Novich  
<emhn@usb.ve>

Universidad “Simón Bolívar”

Copyright ©2010-2013

# Monad Writer

## Ambiente de c  puto con acumulador

- Los Monad que conocemos hasta ahora nos permiten construir un entorno de c  puto para secuenciar operaciones.
-   Y si quiero *acumular* informaci  n a medida que progreso?
  - No se trata de *m  ltiples* resultados – adem  s del valor arbitrario objetivo del c  culo, tenemos otra colecci  n de resultados o informaci  n intermedia relevante.
  - Esa colecci  n *s  lo* puede crecer en tama  o.
  - Como la bit  cora de eventos de una aplicaci  n.



# Monad Writer

## Ambiente de c  puto con acumulador

- Los Monad que conocemos hasta ahora nos permiten construir un entorno de c  puto para secuenciar operaciones.
-   Y si quiero *acumular* informaci  n a medida que progreso?
  - No se trata de *m  ltiples* resultados – adem  s del valor arbitrario objetivo del c  lculo, tenemos otro valor colecci  n de resultados o informaci  n intermedia relevante.
  - Esa colecci  n *s  lo* puede crecer en tama  o.
  - Como la bit  cora de eventos de una aplicaci  n.
- Con esas premisas, el Monad deber  a
  - Operar sobre un valor y resultados acumulados –  $(a, [c])$ .
  - Para transformar el valor (computar) –  $(a \rightarrow (b, [c]))$ .
  - Preservar el resultado y los acumulados –  $(b, [c])$ .



# Monad Writer

## Ambiente de cómputo con acumulador

- Los Monad que conocemos hasta ahora nos permiten construir un entorno de cómputo para secuenciar operaciones.
- ¿Y si quiero *acumular* información a medida que progreso?
  - No se trata de *múltiples* resultados – además del valor arbitrario objetivo del cálculo, tenemos otra colección de resultados o información intermedia relevante.
  - Esa colección *sólo* puede crecer en tamaño.
  - Como la bitácora de eventos de una aplicación.
- Con esas premisas, el Monad debería
  - Operar sobre un valor y resultados acumulados –  $(a, [c])$ .
  - Para transformar el valor (computar) –  $(a \rightarrow (b, [c]))$ .
  - Preservar el resultado y los acumulados –  $(b, [c])$ .

¡Generalizable si usamos Monoid en lugar de  $[a]$ !



# Monad Writer

## Ambiente de c  puto con acumulador

- Comencemos por definir un tipo para el c  culo

```
newtype Writer w a = Writer { runWriter :: (a, w) }
```

- El Monad es `Writer w` – polim  rfico en `a`
  - `a` es el tipo de datos para el c  puto en secuencia.
  - `w` es el tipo de datos acumulador de informaci  n.
- El envoltorio de costumbre con su funci  n de extracci  n.



# Monad Writer

## Ambiente de cómputo con acumulador

- Instancia lo más general posible

```
instance (Monoid w) => Monad (Writer w) where
```



# Monad Writer

## Ambiente de c  puto con acumulador

- Instancia lo m  s general posible

```
instance (Monoid w) => Monad (Writer w) where
```

- Inyectar un valor – tupla con el valor y acumulador vac  o

```
return :: a -> Writer w a  
return x = Writer (x, mempty)
```



# Monad Writer

## Ambiente de c  puto con acumulador

- Instancia lo m  s general posible

```
instance (Monoid w) => Monad (Writer w) where
```

- Inyectar un valor – tupla con el valor y acumulador vac  o

```
return :: a -> Writer w a  
return x = Writer (x, mempty)
```

- Combinar un c  puto – calcular el nuevo valor y acumular

```
(>>=) :: Writer w a -> (a -> Writer w b)  
      -> Writer w b  
(Writer (x,v)) >>= f =  
    let (Writer (y, v')) = f x  
    in Writer (y, v 'mappend' v')
```





# Monad Writer

##   C  mo aprovecharlo?

```
import qualified Data.Sequence as DS
import Control.Monad.Writer

logIt :: Int -> Writer (DS.Seq String) Int
logIt x = writer (x, DS.singleton $ "Veo " ++ show x)

multWithLog :: Writer (DS.Seq String) Int
multWithLog = do
  a <- logIt 21
  b <- logIt 2
  tell $ DS.singleton "Multiplicando."
  return $ a*b

> runWriter multWithLog
(42,fromList ["Veo 21","Veo 2","Multiplicando."])
```

- tell permite acumular un mensaje destruyendo el valor

# Hay más de una forma de escribir

- `Control.Monad.Writer.Lazy` provee una versión perezosa – `Control.Monad.Writer.Strict` provee una versión estricta.
- Énfasis en manipular la bitácora

```
writer :: MonadWriter w m => (a, w) -> m a
tell   :: MonadWriter w m => w -> m ()
listen :: MonadWriter w m => m a -> m (a,w)
censor :: MonadWriter w m => (w -> w) -> m a -> m a
```



# Estado mutable

¿Y esto no era programación funcional?

- Programación Funcional – Transparencia Referencial
  - No se puede modificar una estructura “en sitio”.
  - Las estructuras deben reconstruirse para representar los cambios, descartando la original cuando ya no es necesaria.



# Estado mutable

¿Y esto no era programación funcional?

- Programación Funcional – Transparencia Referencial
  - No se puede modificar una estructura “en sitio”.
  - Las estructuras deben reconstruirse para representar los cambios, descartando la original cuando ya no es necesaria.
- Existen cálculos basados en un **estado mutable**
  - Hay un estado “inicial”.
  - En cada paso de cálculo se modifica el estado, posiblemente generando un resultado útil adicional.
  - Al terminar el cálculo puede interesar el resultado final, el estado final o ambos.

Eso parece una secuenciación de cálculos...



# Estado Mutable

## Una representación

¿Qué queremos hacer con un estado?

- Establecer un estado inicial.
- Inspeccionar el estado actual.
- Producir un nuevo resultado y un nuevo estado.



# Estado Mutable

## Una representación

¿Qué queremos hacer con un estado?

- Establecer un estado inicial.
- Inspeccionar el estado actual.
- Producir un nuevo resultado y un nuevo estado.

Nuestro modelo

- El estado puede ser cualquier tipo de datos –  $s$
- El resultado puede ser cualquier tipo de datos –  $a$
- Tomar un estado para producir un resultado y un nuevo estado, se modela naturalmente como una función **transformadora de estado**

$s \rightarrow (a, s)$



# Deduciendo el State Monad

Voy a hacer “trampa” porque puedo

- Comencemos por definir un tipo para el transformador

```
type Estado s a = s -> (a, s)
```

- Nuestro monad contendrá una **función** transformadora.
  - $s$  es cualquier tipo de datos que modele el estado.
  - $a$  es cualquier tipo de datos que modele resultados.

El Monad envuelve una *función* –  
esa es la clave para comprender su funcionamiento.



# Deduciendo el State Monad

## Proveer las funciones mínimas

- Inyectar un valor – función que preserve el estado, incluyendo el valor

```
returnE :: a -> Estado s a  
returnE a = \s -> (a,s)
```





# Deduciendo el State Monad

## Proveer las funciones mínimas

- Inyectar un valor – función que preserve el estado, incluyendo el valor

```
returnE :: a -> Estado s a  
returnE a = \s -> (a,s)
```

- Combinar un cómputo – aplicarlo y producir el nuevo estado

```
bindE :: (Estado s a)  
      -> (a -> Estado s b)  
      -> (Estado s b)  
bindE m k = \s -> let (a, s') = m s  
                  in (k a) s'
```

Si sustituimos `Estado s a` por su tipo,  
puede resultar más fácil de comprender ...



# Sustituyendo Estado s a

```
bindE :: (s -> (a,s))  
      -> (a -> s -> (b,s))  
      -> (s -> (b,s))  
  
bindE cambiar crearCambio estadoActual =  
  let (resultado, estadoNuevo) = cambiar estadoActual  
  in  (crearCambio resultado) estadoNuevo
```

Estado s a no es más que una *función*



# ¿Cómo interactúo con el estado?

Como siempre, las definiciones de `return` y `bind` modelan el método de cómputo, escondiendo el procesamiento. Necesitamos funciones auxiliares.

- Obtener el estado actual ...

```
getSt :: Estado s a  
getSt = \s -> (s,s)
```

...es *retornarlo* como si fuera el resultado.



# ¿Cómo interactúo con el estado?

Como siempre, las definiciones de `return` y `bind` modelan el método de cómputo, escondiendo el procesamiento. Necesitamos funciones auxiliares.

- Obtener el estado actual ...

```
getSt :: Estado s a  
getSt = \s -> (s,s)
```

...es *retornarlo* como si fuera el resultado.

- Inyectar un estado particular ...

```
putSt :: s -> Estado s a  
putSt s = \_ -> ((),s)
```

...es *ignorar* el actual y forzar el nuevo.

Hasta aquí llegó mi “trampa”



# Monad State

Will the real State Monad please stand up!

Tenemos que usar un tipo concreto

- Nos obliga a usar un constructor ...
- ... con los consecuentes empaquete y desempaquete
- Más una función de escape

```
newtype State s a = State {  
    runState :: s -> (a,s)  
}  
  
instance Monad (State s) where  
    return a          = State $ \s -> (a,s)  
    (State x) >>= f = State $ \s -> let (v,s') = x s  
                                     in runState (f v) s'
```



# Monad State

Con toda la formalidad del caso

Las funciones auxiliares están en una clase diferente

```
class MonadState m s | m -> s where
    get  :: m s
    put  :: s -> m ()

instance MonadState (State s) s where
    get  = State $ \s -> (s,s)
    put s = State $ \_ -> ((),s)
```

La notación `| m -> s` se lee “donde `m` determina a `s`” y se conoce como **dependencia funcional** (*fundep*).

- Ayudan al sistema de tipos a resolver ambigüedades.
- Hablaremos de ellos más adelante.



# Usando Monad State

## Números pseudo-aleatorios (PRNG)

- Genera una secuencia **finita** de números que tienen propiedades similares a las de números aleatorios verdaderos.
- La secuencia
  - Basada en dos valores fijos – **multiplicador** (primo) y **módulo**.
  - Parte con un valor suministrado por el usuario – la **semilla**.



# Usando Monad State

## Números pseudo-aleatorios (PRNG)

- Genera una secuencia **finita** de números que tienen propiedades similares a las de números aleatorios verdaderos.
- La secuencia
  - Basada en dos valores fijos – **multiplicador** (primo) y **módulo**.
  - Parte con un valor suministrado por el usuario – la **semilla**.
- Algebra de congruencias – usar la semilla para producir el siguiente número de la secuencia, y la nueva semilla.

$$semilla_{n+1} = (semilla_n \times multiplicador) \equiv modulo$$

$$azar_{n+1} = \frac{semilla_{n+1}}{modulo}$$





# Usando Monad State

## Números pseudo-aleatorios – El estado

- Definiremos el estado representando el cambio en la semilla

```
newtype PRNG a = PRNG (Integer -> (a, Integer))
```



# Usando Monad State

## Números pseudo-aleatorios – El estado

- Definiremos el estado representando el cambio en la semilla

```
newtype PRNG a = PRNG (Integer -> (a, Integer))
```

- Necesitamos una función para avanzar la secuencia

```
rng :: Integer -> (Float,Integer)
rng actual = ( fromInteger (nueva) /
               fromInteger(modulo),
               nueva )

where
    nueva          = (actual * multiplicador)
                     `mod` modulo
    multiplicador = 1812433253
    modulo         = 2^32
```



# Usando Monad State

## Números pseudo-aleatorios – La plomería

- La instancia de Monad

```
instance Monad PRNG where
  return x = PRNG $ \semilla -> (x, semilla)

  (PRNG g0) >>= f = PRNG $
    \semilla -> let (y, semilla') = g0 semilla
                  (PRNG g1) = f y
                  in g1 semilla'
```



# Usando Monad State

## Números pseudo-aleatorios – La plomería

- La instancia de Monad

```
instance Monad PRNG where
  return x = PRNG $ \semilla -> (x, semilla)

  (PRNG g0) >>= f = PRNG $
    \semilla -> let (y, semilla') = g0 semilla
                  (PRNG g1) = f y
                  in g1 semilla'
```

- Funciones para comodidad

```
initialSeed      = 3121281023
random           = PRNG rng
generate (PRNG f) = (fst.f) initialSeed
```



# Usando Monad State

## Números pseudo-aleatorios – Uso

```
pareja = do
  a <- random
  b <- random
  return (a,b)

lista n = do
  x <- random
  xs <- lista $ n-1
  return $ if (n == 0) then [] else (x:xs)

> generate pareja
(0.628972,0.3711084)
> generate $ lista 5
[0.628972,0.3711084,0.18037394,0.95906913,0.55127066]
```



# Pero se puede hacer con menos código

Control.Monad.State es tu amigo

```
import Control.Monad.State

random :: State Integer Float
random = do
  actual <- get
  let (azar,nueva) = rng actual
  put nueva
  return azar

initialState = 3121281023
generate rs = evalState rs initialState
```

- No hace falta el newtype PRNG ni escribir la instancia.
- Basta escribir firmas correctas para los transformadores.
- Todo lo demás funciona exactamente igual.

# Y puede ahorrar *mucho* tiempo

Si sabes lo que estás haciendo...

- El secreto es identificar los tipos para estado y valor – supongamos una pila simulada con una lista
  - El estado – la pila como un todo.
  - El valor – lo que se obtiene al hacer pop.
- Esto es todo lo que hace falta...

```
type Stack a = [a]

pop :: State (Stack a) a
pop = State $ \(x:xs) -> (x,xs)

push :: a -> State (Stack a) ()
push a = State $ \(xs) -> ((),a:xs)
```

¡Pilas monádicas genéricas!



# Pilas monádicas genéricas...

```
> runState (push 3 >> push 2 >> pop >> push 1) []  
((), [1, 3])  
> runState (push 'o' >>  
            pop >>= \x -> push x >> push x >>  
            push 'f') []  
((), "foo")  
> runState (push "foo" >> push "bar" >>  
            put ["wtf?"]) []  
((), ["wtf?"])  
> runState (push "foo" >> push "bar" >>  
            put ["wtf?"] >> pop) []  
("wtf?", [])
```

Not bad!



## Y como diseñamos la pila *genérica*...

```
> let f = (sequence $ replicate 2 (push 'o')) >>
    push 'f'
> let q = push 'x' >>
    (sequence $ replicate 2 (push 'u'))
    >> push 'q'
> evalState ((push (runState f [])) >>
    (push (runState q [])) >> get)
[]
[((()), "quux"), ((()), "foo")]
```

Una pila de pilas...



# Hay más de una forma de manipular el estado

- `Control.Monad.State.Lazy` provee una versión perezosa – `Control.Monad.State.Strict` provee una versión estricta.
- Enfasis en manipular el estado

```
get      :: MonadState s m => m s
put      :: MonadState s m => s -> m ()
modify   :: MonadState s m => (s -> s) -> m ()
gets     :: MonadState s m => (s -> a) -> m a
```

- Ejecutar las transformaciones y acceder al resultado

```
runState  :: State s a -> s -> (a,s)
evalState :: State s a -> s -> a
execState :: State s a -> s -> s
```



# Monad IO

Gracias Wadler por los favores concedidos

- `type IO = World -> (a,World)` – No es Haskell válido :-)
- Es un Monad State – para alguna definición “razonable” de `World`.
- Programa – cómputo monádico en este Monad State especializado

```
main :: IO ()
```

- Por definición de `>>=`, el mundo se mueve en una sola “dirección” – no hay copias ni viaje en el tiempo.



# Aprovechando el Monad IO

- No hay forma de escapar del Monad IO – obviamente opaco.
- Pero los cálculos en el Monad IO son objetos de primera clase
  - *Construirlos* es diferente a *ejecutarlos*.
  - Pueden combinarse con el resto de los tipos de datos

```
> let msgs = [ putStr "Hello", putStr " World!\n" ]
> :type msgs
msgs :: [IO ()]
> sequence_ msgs
Hello World!
```

- Toda interacción con el mundo se encapsula en el Monad IO
  - En el ambiente de ejecución de GHC.
  - En alguna librería externa “ligada” a Haskell (FFI).

# Más combinadores sobre Monads

## Listas de Monads

```
sequence :: (Monad m) => [m a] -> m [a]
sequence = foldr mcons (return [])
  where
    mcons p q = p >>=
      \x -> q >>=
      \y -> return (x : y)

> sequence [Just 1, Just 2, Just 3]
Just [1,2,3]
> sequence [print 1, print 2, print 3]
1
2
3
[(), (), ()]
```

Lista de Monad se convierte en Monad con lista de resultados.



UNIVERSIDAD SIMÓN BOLÍVAR

# Más combinadores sobre Monads

## Listas de Monads

```
sequence_ :: (Monad m) => [m a] -> m ()  
sequence_ = foldr (>>) (return ())
```

```
> sequence_ [Just 1, Just 2, Just 3]  
Just ()  
> sequence_ [print 1, print 2, print 3]  
1  
2  
3
```

Lista de Monad se convierte en Monad conteniendo “void”,  
pero los efectos de borde ocurren.

# Más combinadores sobre Monads

## Listas de Monads – filter monádico

```
filterM :: Monad m => (a -> m Bool) -> [a] -> m [a]
filterM p []          = return []
filterM p (x : xs) =
  do
    b  <- p x
    ys <- filterM p xs
    return (if b then (x : ys) else ys)

> filterM (\x -> Just (x>0)) [2,1,0,-1]
Just [2,1]
> filterM (\x -> [x>0]) [2,1,0,-1]
[[2,1]]
```

Lista se convierte en Monad con lista de valores que cumplan un predicado monádico.



# Más combinadores sobre Monads

## Listas de Monads – map monádico

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f as = sequence (map f as)
```

```
> mapM Just [0,1,2]
Just [0,1,2]
> mapM print [0,1,2]
0
1
2
[(),(),()]
```

Lista de valores se convierte en Monad conteniendo la lista de resultados de una función monádica.





# Más combinadores sobre Monads

## Listas de Monads – map monádico

```
mapM_ :: Monad m => (a -> m b) -> [a] -> m ()  
mapM_ f as = sequence_ (map f as)
```

```
> mapM_ Just [0,1,2]  
Just ()  
> mapM_ print [0,1,2]  
0  
1  
2
```

Lista de valores se convierte en Monad conteniendo “void” pero los efectos de borde de la función monádica ocurren por *cada* elemento de la lista original.



# Monad IO y combinadores

- `echo` – repetir los argumentos de línea de comandos



# Monad IO y combinadores

- echo – repetir los argumentos de línea de comandos

```
main = (intersperse " ") 'liftM' getArgs >>=
      mapM_ putStr >> putChar '\n'
```

(IO es Functor – cambien 'liftM' por <\$> y verán)

- sort – ordenar la entrada



# Monad IO y combinadores

- echo – repetir los argumentos de línea de comandos

```
main = (intersperse " ") 'liftM' getArgs >>=
      mapM_ putStr >> putChar '\n'
```

(IO es Functor – cambien 'liftM' por <\$> y verán)

- sort – ordenar la entrada

```
main = (mapM_ putStrLn . sort . lines) =<<
      getContents
```



# Monad Reader

## Ambiente constante “sólo lectura”

- ¿Y si el estado debe ser **immutable** parcial o totalmente?
  - La configuración de una aplicación.
  - Monad State sirve, pero no impide cambios “accidentales”.



# Monad Reader

## Ambiente constante “sólo lectura”

- ¿Y si el estado debe ser **immutable** parcial o totalmente?
  - La configuración de una aplicación.
  - Monad State sirve, pero no impide cambios “accidentales”.
- Si se tratara de una función sería  $e \rightarrow a \dots$ 
  - $\dots$  recibiría un “ambiente” – e por *environment*.
  - $\dots$  para calcular algún resultado – a arbitrario.
- El cómputo debe aprovechar el ambiente
  - Para obtener valores del ambiente – ask
  - Debe dejarlo disponible para el *próximo* cómputo en la secuencia.

Se repite la receta de State.



# Monad Reader

## Cómputos sobre un ambiente constante

```
newtype Reader e a = Reader { runReader :: (e -> a) }

instance Monad (Reader e) where
    return a          = Reader $ \e -> a
    (Reader r) >>= f = Reader $ \e -> f (r e) e

class MonadReader e m | m -> e where
    ask    :: m e
    local  :: (e -> e) -> m a -> m a

instance MonadReader (Reader e) where
    ask      = Reader id
    local f c = Reader $ \e -> runReader c (f e)

asks :: (MonadReader e m) => (e -> a) -> m a
asks sel = ask >>= return . sel
```

#  No entiendo >>=!

Lo not  ... A todos nos pasa

El inyector es bastante obvio, pero *bind* luce “raro” –  
veamos que no es para nada raro, paso a paso

```
(Reader r) >>= f = Reader $ \e -> f (r e) e
```





# ¡No entiendo $\gg=$ !

Lo noté ... A todos nos pasa

El inyector es bastante obvio, pero *bind* luce “raro” –  
veamos que no es para nada raro, paso a paso

```
(Reader r) >>= f = Reader $ \e -> f (r e) e
```

- *r* está dentro del Reader – es una función (*e*  $\rightarrow$  *a*)



# ¡No entiendo $\gg=$ !

Lo noté ... A todos nos pasa

El inyector es bastante obvio, pero *bind* luce “raro” –  
veamos que no es para nada raro, paso a paso

```
(Reader r) >>= f = Reader $ \e -> f (r e) e
```

- $r$  está dentro del Reader – es una función  $(e \rightarrow a)$
- Por tanto  $(r\ e)$  no es más que aplicarla – produce algo del tipo  $a$ .



# ¡No entiendo >>=!

Lo noté ... A todos nos pasa

El inyector es bastante obvio, pero *bind* luce “raro” –  
veamos que no es para nada raro, paso a paso

```
(Reader r) >>= f = Reader $ \e -> f (r e) e
```

- *r* está dentro del Reader – es una función (*e* -> *a*)
- Por tanto (*r e*) no es más que aplicarla – produce algo del tipo *a*.
- *f* es una función (*a* -> Reader *e* *a*) – se aplica sobre (*r e*).



# ¡No entiendo >>=!

Lo noté ... A todos nos pasa

El inyector es bastante obvio, pero *bind* luce “raro” –  
veamos que no es para nada raro, paso a paso

```
(Reader r) >>= f = Reader $ \e -> f (r e) e
```

- *r* está dentro del Reader – es una función (*e* -> *a*)
- Por tanto (*r e*) no es más que aplicarla – produce algo del tipo *a*.
- *f* es una función (*a* -> Reader *e a*) – se aplica sobre (*r e*).
- El resultado de *f (r e)* es un Reader *e a* – ¡eso es una función (*e* -> *a*) otra vez!



# ¡No entiendo $\gg=$ !

Lo noté ... A todos nos pasa

El inyector es bastante obvio, pero *bind* luce “raro” –  
veamos que no es para nada raro, paso a paso

```
(Reader r) >>= f = Reader $ \e -> f (r e) e
```

- $r$  está dentro del Reader – es una función  $(e \rightarrow a)$
- Por tanto  $(r\ e)$  no es más que aplicarla – produce algo del tipo  $a$ .
- $f$  es una función  $(a \rightarrow \text{Reader } e\ a)$  – se aplica sobre  $(r\ e)$ .
- El resultado de  $f\ (r\ e)$  es un  $\text{Reader } e\ a$  – ¡eso es una función  $(e \rightarrow a)$  otra vez!
- Entonces  $(f\ (r\ e))$  se puede aplicar al  $e$

Keep calm and Curry on...



# Monad Reader

## Funciones auxiliares

- ask ofrece acceso al ambiente – en este caso un Int

```
> runReader (ask >>= \x -> return (x*3))  
      14  
42
```



# Monad Reader

## Funciones auxiliares

- ask ofrece acceso al ambiente – en este caso un Int

```
> runReader (ask >=> \x -> return (x*3))  
14  
42
```

- local permite ofrecer definiciones locales sobre el ambiente – piensen “alcance dinámico”

```
> runReader (ask >=>  
    local (+9) \x -> return (x*3))  
14  
69
```

- Noten que el primer argumento de local es una *función*.
- Esa función podría modificar el ambiente “temporalmente”.



# Monad Reader

```
import Data.Maybe
import qualified Data.Map as M
import Control.Monad.Reader

data VarType = TInt | TFloat deriving (Eq)
type Symbols = M.Map String VarType

isInteger :: String -> Reader Symbols Bool
isInteger sym = do
  t <- asks (lookupVar sym)
  return (t == TInt)

lookupVar :: String -> Symbols -> VarType
lookupVar n b = fromJust (M.lookup n b)

table = M.fromList [("foo",TInt), ("bar",TFloat)]
```





# Generalización de Monads

- Prelude ofrece lo mínimo para operar con Monads.
- Aprovechar los módulos específicos.
  - `Data.Maybe` – Monad Maybe
  - `Control.Monad` – Clases de tipos y combinadores generales
  - `Control.Monad.State` – Monad State
  - `Control.Monad.Reader` – Monad Reader
  - `Control.Monad.Writer` – Monad Writer
  - `Control.Monad.RWS` – Monad que combina Reader, Writer y State (esperen una semana antes de verla).
  - `System.IO` – Monad IO para I/O general
  - `System.Random` – Monad para RNG y PRNG.
- ...y muchos más.



# Quiero saber más...

- [Documentación de Control.Monad](#)
- [Documentación de Control.Monad.State](#)
- [Documentación de Control.Monad.Reader](#)
- [Documentación de Control.Monad.Writer](#)
- [Documentación de System.IO](#)
- [Documentación de System.Random](#)

